Randomness of Spritz via DieHarder testing

Răzvan Roşie

Abstract

RC4 is a stream cipher included in the TLS protocol, and widely used for encrypting network traffic during the last decades. **Spritz** is a possible candidate for replacing RC4. Spritz is based on a sponge construction and preserves the byte-oriented behaviour existing in RC4, but introduces an interface that provides encryption, hashing or MAC-generation functionalities.

We present here the results obtained after applying several statistical tests on the keystreams generated by Spritz when used in the cipher mode. Our methodology makes use of 1024 keystreams of 2^{25} bits. The algorithm was tested against the DieHarder test suite. None of the tests failed. Few tests produced weak results that were corrected when the number of samples increased.

Keywords: Spritz, RC4, stream cipher, Dieharder, NIST Statistical Test Suite.

1 Introduction

Symmetric-key cryptography contains two fundamental classes of cryptographic algorithms: **block** ciphers and **stream** ciphers. During the past decade, symmetric block ciphers were intensively studied through the AES-election contest organized by NIST. The winning algorithm is considered secure and has been successfully used in practice. On the other hand, less attention was dedicated to the analysis of stream ciphers. The last contest organized by NIST targeted a different set of cryptographic primitives: hash functions.

Stream ciphers are used in encrypting mobile communications as well as network traffic. Many existing algorithms follow a design of simplicity to favour speed. The only information-theoretically secure cipher, the One Time Pad cipher (OTP), serves as an inspirational model for most stream ciphers: the ciphertext is the result of XOR-ing the plain-text with a random key. Since the OTP is impractical due to the necessary length of the key, various approaches are used by stream ciphers to extend a short key to possibly infinitely long keystreams that are applied to the plain-text.

RC4 is one of the stream ciphers included into the SSL and TLS cipher suites for encrypting Internet traffic, and according to many sources [2, 3], RC4 is one of the most popular choices. Designed in 1987, it was subject to many theoretical and practical attacks; a new proposal to replace RC4 was made by Rivest and Schuldt in 2014 - **Spritz** [1]. The proposed stream cipher is based on a sponge construction, but inherits from the byte-oriented design of RC4.

1.1 Motivation

Few analysis of the algorithm exist [4]. Here we investigate a preliminary aspect of **Spritz**, the randomness of the keystreams that are generated, independently by the tests that were used by the architects. In general, a suspect result obtained when testing the randomness of the output generated by a stream cipher or a hash function does not imply the existence of an easily exploitable structural weakness. However, this may jeopardize the chances of the algorithm to be adopted as a standard.

2 Specification of Spritz

As described by its authors in [1], **Spritz** is constructed having in mind the benefits offered by sponge functions [5]. The result is an algorithm that can be used both as a message authentication code, random bit generator or a hash function. However, the primary intention is to use the algorithm and relevant interfaces as a stream cipher.

Spritz follows the standard approach, where the key is extended and added to the plain-text in order to obtain the cipher-text. A *state* of the algorithm consists of the values of six registers i, j, k, w, z, a and the permutation S of $\{0, 1, .., N-1\}$. The default size of S is $N = 256$. As for sponge constructs, the output is generated by a **Squeeze** procedure, which repeatedly calls a pseudo-random function - **Drip** - that updates the state (changes the values of i, j, k registers and acts on the permutation S) in order to output a byte. When used as a stream cipher, the key is firstly absorbed into the state through blocks of 'nibbles' (half bytes) and executing a **Shuffle** procedure after each absorption. The functionality behind this procedure is aimed to randomize the state. **Shuffle** calls specific procedures that act on the state of the algorithm.

The byte-oriented design the algorithm has the disadvantage of generating the output slower than other word-oriented algorithms, as remarked in [1]. Also, Spritz used in hash mode is slower than the recently elected SHA-3. [1]

[1] The width of the permutation in SHA-3's winning proposal is proportional to the size of the word, thus easily adaptable for different architectures.

The pseudocode of the algorithm [1] and the interface for using it as a stream cipher are provided below:

InitializeState(N)
1: $i = j = k = z = a = 0$
2: $w = 1$
3: **for** $v = 0$ **to** $N - 1$ **do**
4: $S[v] = v$
5: **end for**

Shuffle()
1: Whip($2N$)
2: Crush()
3: Whip($2N$)
4: Crush()
5: Whip($2N$)
6: $a = 0$

Whip(r)
1: **for** $v = 0$ **to** $r - 1$ **do**
2: Update()
3: **end for**
4: w = w+1
5: **do** w = w+1
6: **until** $GCD(w, N) = 1$

Crush()
1: **for** $v = 0$ **to** $\lfloor N/2 \rfloor - 1$ **do**
2: **if** $S[v] > S[N - 1 - v]$ **then**
3: Swap($S[v], S[N - 1 - v]$)
4: **end if**
5: **end for**

Squeeze(r)
1: **if** $a > 0$ **then**
2: Shuffle()
3: **end if**
4: P = Array.New(r)
5: **for** $v = 0$ **to** $r - 1$ **do**
6: $P[v]$ = Drip()
7: **end for**
8: **return** P

Drip()
1: **if** $a > 0$ **then**
2: Shuffle()
3: **end if**
4: Update()
5: **return** Output()

Update()
1: $i = i + w$
2: $j = k + S[j + S[i]]$
3: $k = i + k + S[j]$
4: Swap($S[i], S[j]$)

Output()
1: $z = S[j + S[i + S[z + k]]]$
2: **return** z

Absorb(I)
1: **for** $v = 0$ **to** $I.length - 1$ **do**
2: AbsorbByte($I[v]$)
3: **end for**

AbsorbByte(b)
1: AbsorbNibble(LOW(b))
2: AbsorbNibble(HIGH(b))

AbsorbNibble(x)
1: **if** $a = \lfloor N/2 \rfloor$ **then**
2: Shuffle()
3: **end if**
4: Swap($S[a], S[\lfloor N/2 \rfloor + x]$)
5: $a = a + 1$

AbsorbStop()
1: **if** $a = \lfloor N/2 \rfloor$ **then**
2: Shuffle()
3: **end if**
4: $a = a + 1$

The following functions use the sponge to provide hashing, encryption or decryption functionalities.

Encrypt(K, M)
1: KeySetup(K)
2: $C = M+$ Squeeze($M.length$)
3: **return** C

Decrypt(K, C)
1: KeySetup(K)
2: $M = C-$ Squeeze($C.length$)
3: **return** M

Hash(M, r)
1: InitializeState()
2: Absorb(M)
3: AbsorbStop()
4: Absorb(r)
5: **return** Squeeze(r)

EncryptWithIV(K, IV, M)
1: KeySetup(K)
2: AbsorbStop()
3: Absorb(IV)
4: $C = M+$ Squeeze($M.length$)
5: **return** C

KeySetup(K)
1: InitializeState()
2: Absorb(K)

3 Statistical testing of output generated by Spritz

According to [1, 4], the algorithm went through extensive statistical testing. The design rationale is detailed in the specification (including the possible candidates and the choices made for *Update* and *Output*). However, the results of the statistical randomness tests applied on the keystreams generated by Spritz were not included in the paper. The randomness of the output is a key feature every stream cipher must have; thus we expect Spritz to pass all tests, and confirm the claims made by the authors. In this section, we present the methodology and the results obtained by inspecting the keystreams using the **DieHarder** suite [6].

3.1 Methodology

Spritz was used in the stream-cipher mode. In order to obtain the output for feeding DieHarder, we initially generated the keystreams needed by the Encrypt procedure (without initial values). We used randomly generated keys of 32 bytes length. The size of the permutation was 256 bytes (default value). The data generated consisted of 1024 keystreams of 2^{25} bits. The test environment used was a Linux AMI micro-instance, available from Amazon Web Services.

We motivate our choice for using the DieHarder test suite, due to the fact that it includes most of the tests existing in NIST's Statistical Test Suite [7] and the DieHard battery of tests [8]. Also, the test suite allows to keep testing the data until a result is reached with high confidence.

3.2 Tests used in the statistical analysis

DieHarder is a comprehensive tool in terms of the statistical tests that are incorporated. The recent versions fully include the NIST suite consisting of tests for measuring the frequency, block frequency, entropy, runs, matrix rank, longest run, overlapping or non- overlapping template matchings, linear complexity, serial cumulative sums, random excursions and variants. We do not provide the description of these tests. However we provide the description of particular tests from the DieHarder randomness battery that revealed a weak behaviour for smaller number of samples: the Monobit, Serial, RGB Bit Distibution and RGB Permutation tests:

The Monobit Test

> "Counts the 1 bits in a long string of random uints. Compares to expected number, generates a p-value directly from erfc(). Very effective at revealing overtly weak generators; Not so good at determining where stronger ones eventually fail." [6]

Table 1: Results for 1024 keystreams of length 2^{25}.

Test Name	tuple	psample	p-value	Result
diehard_birthdays	0	100	0.63054096	Passed
diehard_operm5	0	100	0.1282933	Passed
diehard_rank˙32x32	0	100	0.41084472	Passed
diehard_rank˙6x8	0	100	0.29555723	Passed
diehard_bitstream	0	100	0.22681182	Passed
diehard_opso	0	100	0.95338252	Passed
diehard_oqso	0	100	0.08739302	Passed
diehard_dna	0	100	0.88528270	Passed
diehard_count_1s_str	0	100	0.17277707	Passed
diehard_count_1s_byt	0	100	0.68141261	Passed
diehard_parking_lot	0	100	0.08478625	Passed
diehard_2d_sphere	2	100	0.81343673	Passed
diehard_3d_sphere	3	100	0.84124724	Passed
diehard_squeeze	0	100	0.59072752	Passed
diehard_sums	0	100	0.24236920	Passed
sts_monobit	0	100	0.99792691	Weak
sts_monobit	0	200	0.76653420	Passed
sts_runs	0	100	0.97730946	Passed
sts_serial	1	100	0.99792691	Weak
sts_serial	2	100	0.89538406	Passed
sts_serial	3	100	0.62255692	Passed
sts_serial	3	100	0.97791641	Passed
sts_serial	4	100	0.98491218	Passed
sts_serial	4	100	0.72518227	Passed
rgb_bitdist	1	100	0.97541932	Passed
rgb_bitdist	2	100	0.99908738	Weak
rgb_bitdist	2	200	0.99908738	Passed
rgb_bitdist	3	100	0.60061288	Passed
rgb_bitdist	4	100	0.95070961	Passed
rgb_bitdist	5	100	0.970557854	Passed
rgb_permutations	2	100	0.99583035	Weak
rgb_permutations	2	200	0.49210341	Passed
rgb_permutations	3	100	0.34685706	Passed
rgb_permutations	4	100	0.10607070	Passed
rgb_permutations	5	100	0.47826848	Passed
rgb_lagged_sum	0	100	0.48016287	Passed
rgb_lagged_sum	1	100	0.05039327	Passed
rgb_lagged_sum	2	100	0.10018457	Passed

The Serial Test

"Accumulates the frequencies of overlapping n-tuples of bits drawn from a source of random integers. The expected distribution of n-bit patterns is multinomial with $p = 2^{-n}$ e.g. the four 2-bit patterns 00 01 10 11 should occur with equal probability." [6]

The RGB Bit Distribution Test

"Accumulates the frequencies of all n-tuples of bits in a list of random integers and compares the distribution thus generated with the theoretical (binomial) histogram, forming chisq and the associated p-value. In this test n-tuples are selected without overlap (e.g. 01—10—10—01—11—00—01—10) so the samples are independent." [6]

The RGB Permutation Test

"This is a non-overlapping test that simply counts order permutations of random numbers, pulled out n at a time. There are n! permutations and all are equally likely. The samples are independent, so one can do a simple chisq test on the count vector with n! - 1 degrees of freedom." [6]

3.3 Results

The results obtained after running the DieHarder tests confirm the claims of the specification of the algorithm. The *p*-values obtained from the tests were all greater than 0.99, except for the Monobit, Serial, RGB Bit Distribution and RGB Permutation tests. When the number of *p-samples* was increased to 200, the previous tests passed.

Also, a close to 0.01 value was identified for several other tests. To improve the accuracy of the results a larger amount of data may be needed and the number of *p-samples* may be increased as well.

4 Conclusion

This report investigates the randomness of the output generated by Spritz through statistical means. We applied the DieHarder test battery over a set of keystreams produced by the algorithm. The results do not indicate any failure of a randomness statistical test. Suspect *p-values* are observed for few tests. This behaviour is completely eliminated when DieHarder increases the number of samples and repeats the tests - a fact confirms the initial expectations - no non-random behaviour was observed in Spritz.

4.1 Future Work

Spritz is an interesting framework, given the large range of possible applications. As mentioned in the proposal, additional investigations are needed for Spritz operating as a hash function, deterministic random bit generator or message authentication code provider. A future investigation can analyze the algorithm statistically and structurally when used in these modes.

Other improvements can be summarized in testing larger input files or using different statistical tests, that are not available in DieHarder. Many of these tests may be performed in a parallel way. An existing work [9] provides a testing methodology for hash functions. In future, this can be used to investigate the randomness of Spritz when used in the hash mode.

References

[1] Spritz - a spongy RC4-like stream cipher and hash function, *R.L. Rivest, J. C. N. Schuldt*, 2014

[2] An Effective RC4 Stream Cipher, *T.D.B Weerasinghe*, 2014

[3] Prohibiting RC4 Cipher Suites in TLS, *A. Popov*, http://www.ietf.org/proceedings/89/slides/slides-89-uta-3.pdf

[4] Statistical weakness in Spritz against VMPC-R: in search for the RC4 replacement, *B. Zoltak*, 2014

[5] Cryptographic sponge functions, *G. Bertoni, J. Daemen, M. Peeters and G. Van Assche*, 2011

[6] The DieHarder Test Suite, www.phy.duke.edu/~rgb/General/dieharder.php

[7] NIST's Statistical Test Suite, http://csrc.nist.gov/groups/ST/toolkit/rng/stat-stests.html

[8] The Diehard Battery of Tests of Randomness, http://stat.fsu.edu/pub/diehard/

[9] GPU Parallel Statistical and Cube Test Analysis of the SHA-3 Finalist Candidate Hash Functions, *Alan Kaminsky*, 2011.